Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 13 (6.6.2018)

Graphenalgorithmen II

UNI FREIBURG

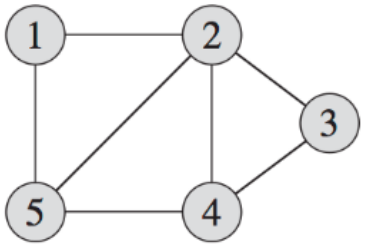
Yannic Maus

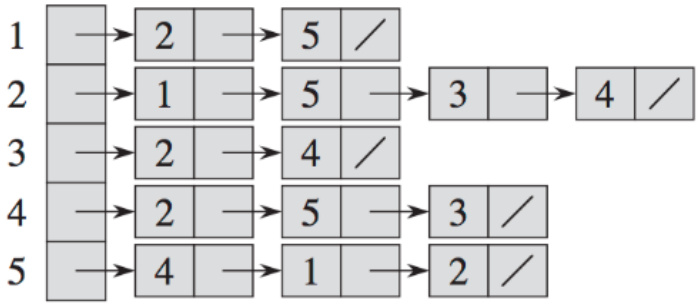
Algorithmen und Komplexität

Repräsentation von Graphen

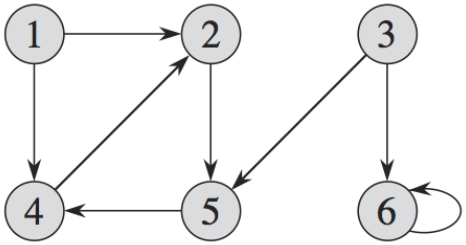
Zwei klassische Arten, einen Graphen im Rechner zu repräsentieren

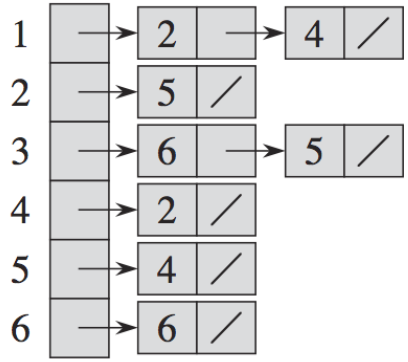
- **Adjazenzmatrix:** Platzverbrauch $O(|V|^2) = O(n^2)$
- **Adjazenzlisten:** Platzverbrauch $O(|V| + |E|) = O(n + m)$





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0





	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Breitensuche von allgemeinen Graphen

UNI FREIBURG

Hauptunterschied Binärbaum T $\Leftrightarrow$ allg. Graph G

- Graph G kann Zyklen haben

Breitensuche in Graph G (Start bei Knoten $s \in V$)

- **Zyklen: markiere** Knoten, welche man schon gesehen hat
- **Markiere** Knoten s, hänge s in die Queue
- Wie bisher, nehme immer den ersten Knoten u aus der Queue:
  - **besuche Knoten u**
  - Gehe durch die Nachbarn v von u
    Falls v nicht markiert, markiere v und hänge v in Queue
    Falls v markiert ist, muss nichts getan werden

BFS Baum: Pseudocode

- Wir merken uns zusätzlich die Distanz zu s im Baum

```
BFS-Tree(s):
    Q = new Queue();
    for all u in V: u.marked = false;
    s.marked = true;
    s.parent = NULL;
    s.d = 0
    Q.enqueue(s)
    while not Q.empty() do
        u = Q.dequeue()
        visit(u)
        for v in u.neighbors do
            if not v.marked then
                v.marked = true;
                v.parent = u;
                v.d = u.d + 1;
                Q.enqueue(v)
```

Analyse Breitensuche

Im BFS-Baum eines ungewichteten Graphen ist die Distanz von jedem Knoten u zur Wurzel s gleich $d_G(s,u)$.

- Die ungewichteten Abstände zwischen allen Knotenpaaren lassen sich in Zeit $O\big(|V| \cdot (|V| + |E|)\big)$ berechnen
- Jeder Graph lässt sich in Zeit $O(|V| + |E|)$ auf Zusammenhang testen
- Die Anzahl der Zusammenhangskomponenten lässt sich in Zeit $O(|V| + |E|)$ berechnen
- Ein ungerichteter Graph lässt sich in Zeit $O(|V| + |E|)$ auf Kreisfreiheit testen (dies geht auch in $O(|V|)$, Übung?)

Tiefensuche in allgemeinen Graphen

UNI FREIBURG

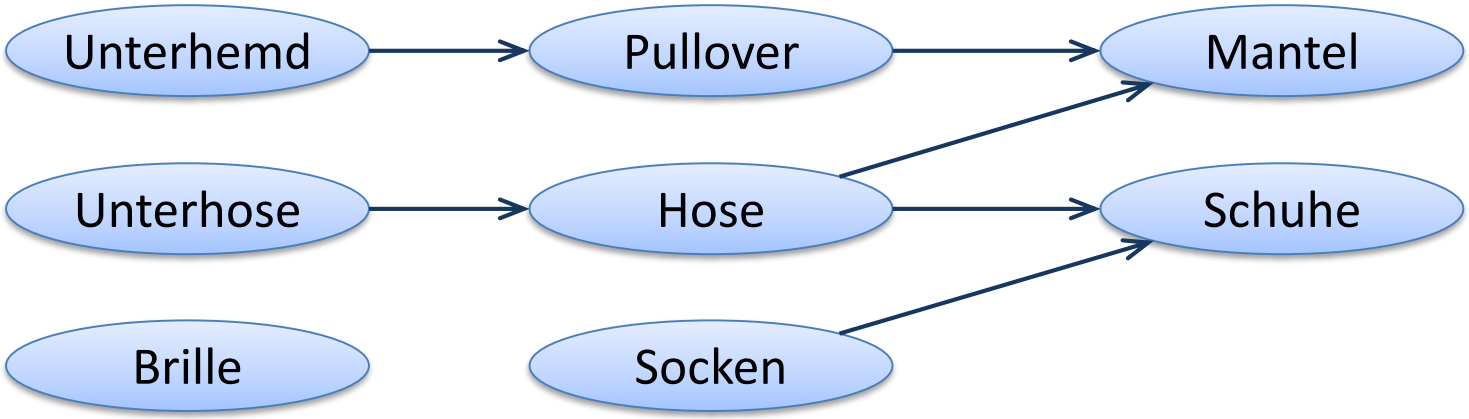
Grundidee Tiefensuche in G (Start bei Knoten $s \in V$)

- **Markiere Knoten v** (am Anfang ist $v = s$)
- Besuche die Nachbarn von v der Reihe nach *rekursiv*
- **Nachdem** alle Nachbarn besucht sind, **besuche s**

- **rekursiv:** Beim Besuchen der Nachbarn werden deren Nachbarn besucht, und dabei deren Nachbarn, etc.
- **Zyklen in G:** Besuche jeweils nur Knoten, welche noch nicht markiert sind

Warum schauen wir DFS so intensiv an?

- Wird sehr häufig als Subroutine verwendet
- Impliziert einen (einfachen) Algorithmus um **Zykel (Kreise)** in **gerichteten** Graphen zu finden
- Ermöglicht einen Algorithmus zum **topologischen Sortieren.**



Ausgabe: Möglich Anziehreihenfolge, wichtig beim Kompilieren

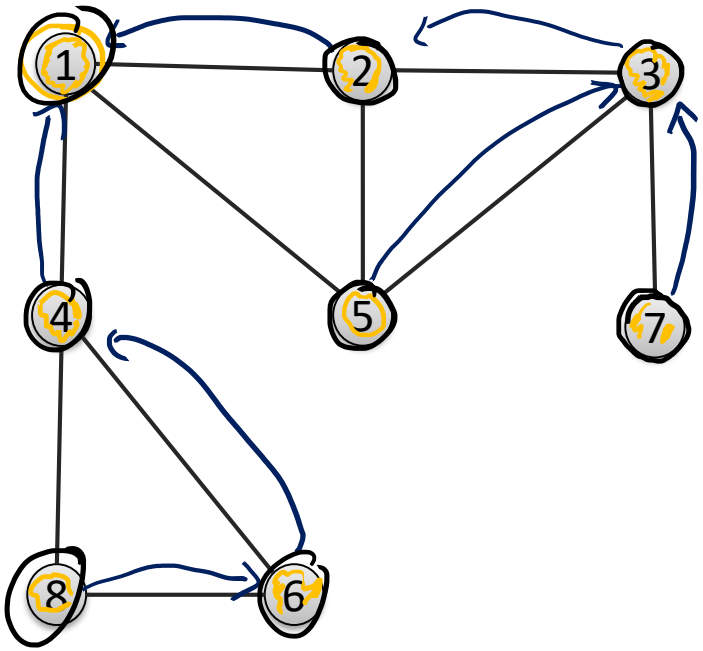
- Viele weitere Anwendungen: Zusammenhangskomponenten finden, Artikulationsknoten finden, Bidirected Components finden, usw.

Tiefensuche: Pseudocode

```
DFS-Traversal(s):
    for all u in V: u.color = white;
    DFS-visit(s, NULL)

DFS-visit(u, p):
    u.color = gray;
    u.parent = p;
    for all v in u.neighbors do
      if v.color = white
        DFS-visit(v, u)
    visit node u;
    u.color = black;
```

Tiefensuche: Beispiel



Tiefensuche: Analyse

In der gleichen Art wie bei der Breitensuche, kann man auch bei der Tiefensuche einen Spannbaum konstruieren

- Die Eigenschaften dieses DFS-Baums werden wir noch anschauen

Die Laufzeit der Tiefensuche (DFS-Traversierung) ist $O(n+m)$.

- Wir färben die Knoten weiß, grau und schwarz wie vorher
  - nicht markiert = weiß, markiert = grau, besucht = schwarz

$\sum_{v \in V} Adj(v) = O(m)$

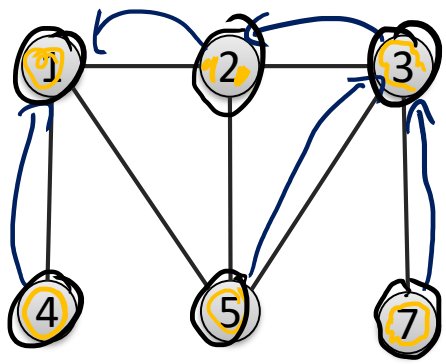
DFS- “Klammer”-Theorem

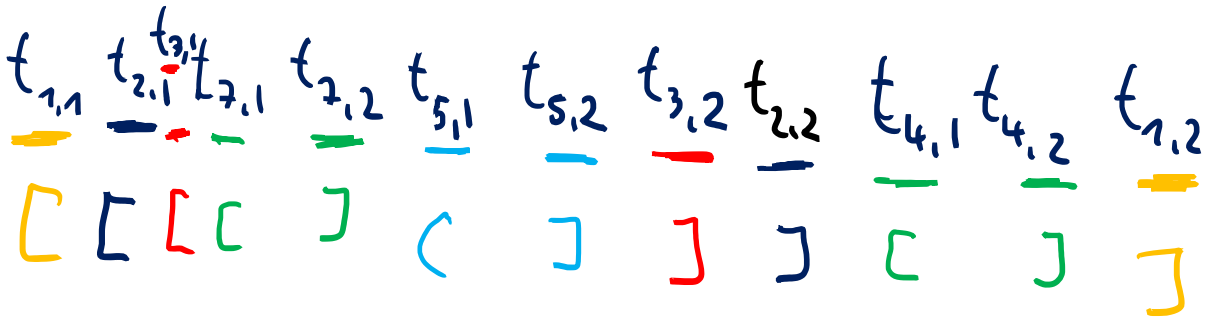
Wir definieren für jeden Knoten v die folgenden zwei Zeitpunkte

- $t_{v,1}$: Zeitpunkt, wenn v in der DFS-Suche grau gefärbt wird
- $t_{v,2}$: Zeitpunkt, wenn v in der DFS-Suche schwarz gefärbt wird

Theorem: Im DFS-Baum ist ein Knoten v genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.

Beispiel:





DFS- “Klammer”-Theorem:

Theorem: Im DFS-Baum ist ein Knoten v ist genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.

Implikationen

- Zwei Intervalle sind entweder disjunkt, oder das eine ist komplett im anderen enthalten.

- Wieso “Klammer”-Theorem:
  Wenn man bei jedem $t_{v,1}$ eine öffnende Klammer und bei jedem $t_{v,2}$ eine schließende Klammer hinschreibt, bekommt man ein Klammerausdruck, welcher korrekt geschachtelt ist.

- Ein weisser Knoten v, welcher in der rekursiven Suche von u entdeckt wird, wird schwarz, bevor die Rekursion zu u zurückkehrt.

DFS- “Klammer”-Theorem: Warum?

Theorem: Im DFS-Baum ist ein Knoten v ist genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.

Wieso ist dies nützlich?

- Erhöht das Verständnis für den resultierenden DFS-Baum
- Wir benötigen das Theorem z.B. bei der Korrektheit des Algorithmus’ zur topologischen Sortierung

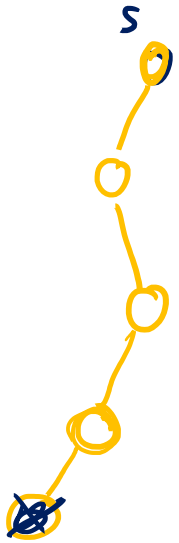
DFS- “Klammer”-Theorem

Theorem: Im DFS-Baum ist ein Knoten v ist genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.

- Insbesondere sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

Beweis:

- graue Knoten bilden immer einen Pfad (G zusammenhängend)



Was kann passieren

weiß → grau hängen einen Knoten an den Pfad an

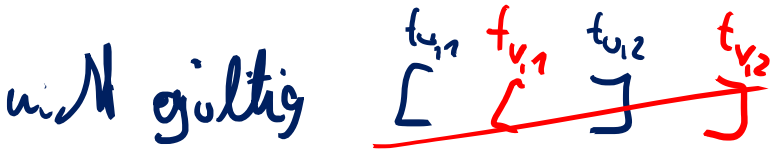
grau → schwarz Pfad wird nicht um ein verkürzt

Pfad ändert sich nur ganz unten

DFS- “Klammer”-Theorem

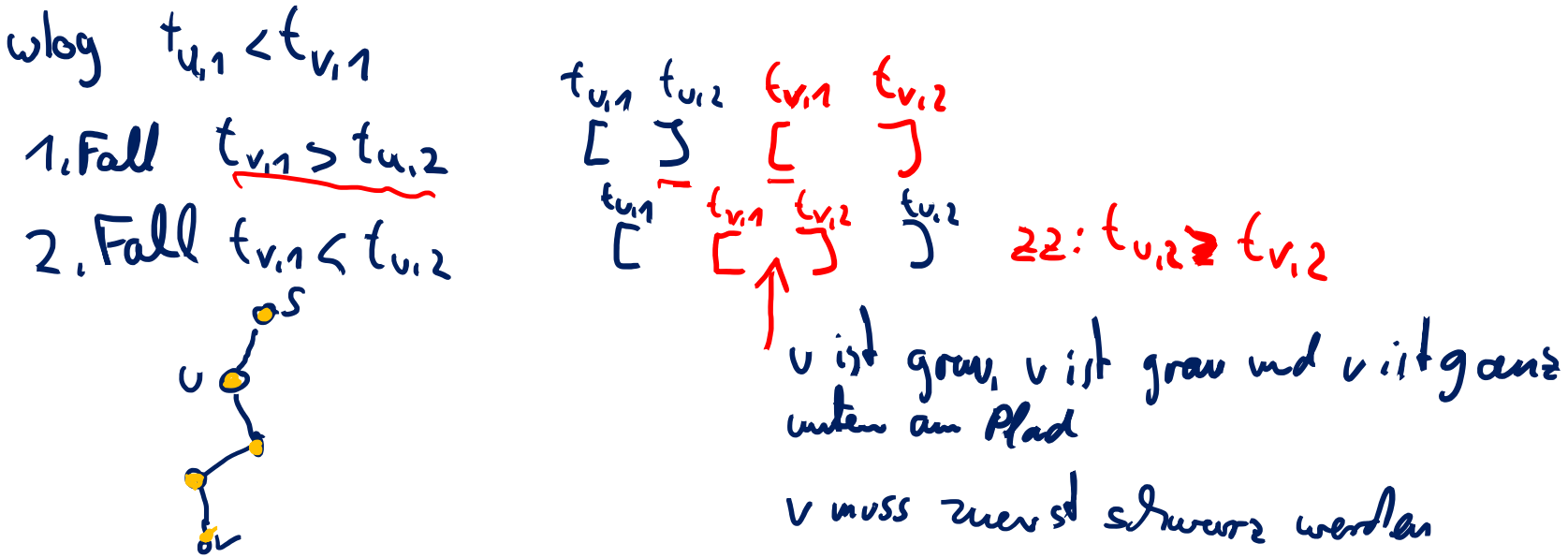
Theorem: Im DFS-Baum ist ein Knoten v ist genau dann im Teilbaum eines Knoten u, falls das Intervall $[t_{v,1}, t_{v,2}]$ vollständig im Intervall $[t_{u,1}, t_{u,2}]$ enthalten ist.

- Insbesondere sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.



Beweis:

- graue Knoten bilden immer einen Pfad (G zusammenhängend)



Weiße Pfade

Theorem: In einem DFS-Baum ist ein Knoten v genau dann im Teilbaum eines Knoten u, falls unmittelbar vor dem Markieren von u, ein komplett weißer Pfad von u nach v besteht.

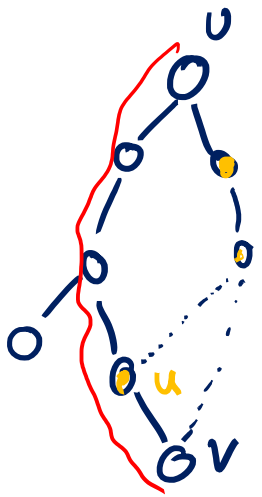
Beweis:

DFS-visit(u)

→ u wird grau

[Nachbarn besucht]

u wird schwarz



notwendig:

Algorithmus läuft nur über weiße Knoten.

Ohne weißen Pfad kann man nie von u zu v kommen.

Klassifizierung der Kanten (bei DFS-Suche)

Baumkanten:

- (u, v) ist eine Baumkante, falls v von u aus entdeckt wird

Rückwärtskanten:

- (u, v) ist eine Rückwärtskante, falls v ein Vorgängerknoten von u ist

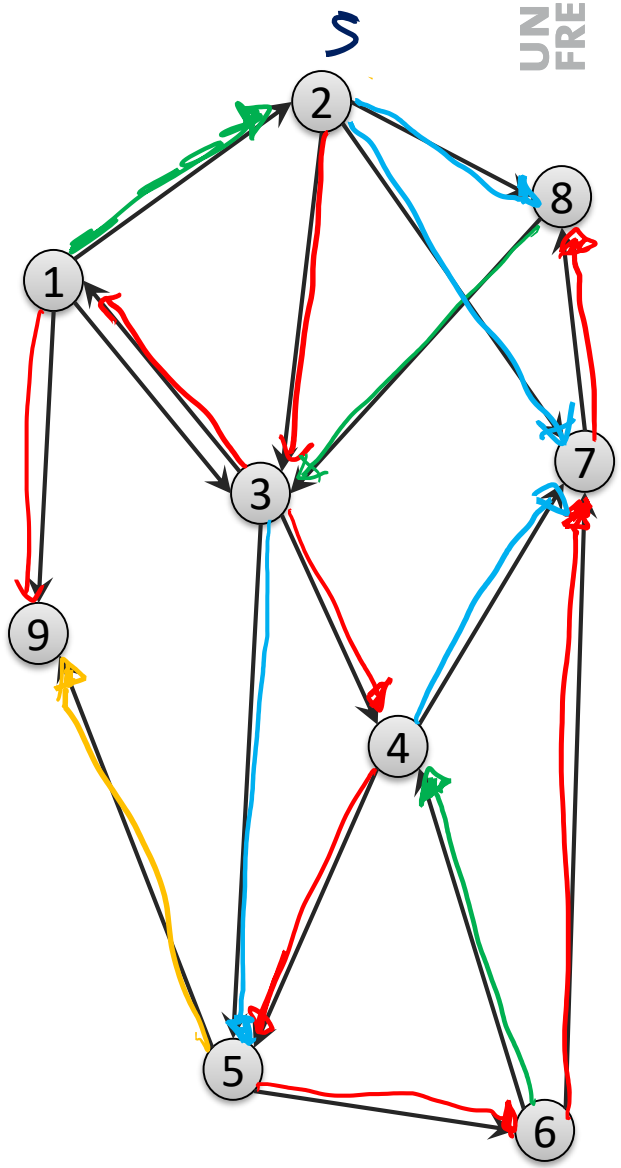
u ist im Teilbaum von v

Vorwärtskanten:

- (u, v) ist eine Vorwärtskante, falls v ein Nachfolgerknoten von u ist

Querkanten:

- Alle übrigen Kanten



Klassifizierung der Kanten (bei DFS-Suche)

Baumkante (u, v):

falls v weiß ist $\rightarrow (u,v)$ Baumkante

Rückwärtskante (u, v):

falls v grau ist $\rightarrow (u,v)$ Rückwärtskante

Vorwärtskante (u, v):

Querkante (u, v):

v schwarz

Vorwärtskante: $t_{u,1}\ t_{v,1}\ t_{v,2}\ t_{u,2}$ — $[\ [\ ]\ ]$

$t_{v,1}, t_{v,2} > t_{u,1}$

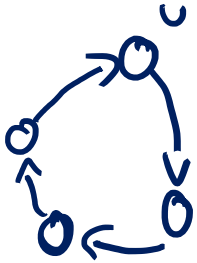
Querkante: $t_{v,1}\ t_{v,2}\ t_{u,1}\ t_{u,2}$ — $[\ ]\ [\ ]$

$t_{v,1}, t_{v,2} < t_{u,1}$

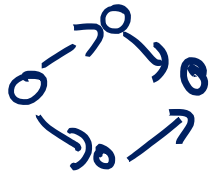
DFS – Gerichtete Graphen

Theorem: Ein gerichteter Graph hat genau dann keine Zyklen, falls es bei der DFS-Suche keine Rückwärtskanten gibt.

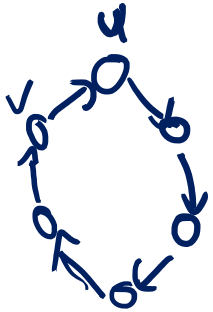
Falls es Rückwärtskanten gibt



✓ ⇒ es gibt einen Kreis



Falls es einen Kreis gibt



Sei u der erste Knoten, den wir besuchen

- haben einen weißen Pfad zu v ⇒ v im Teilbaum von u

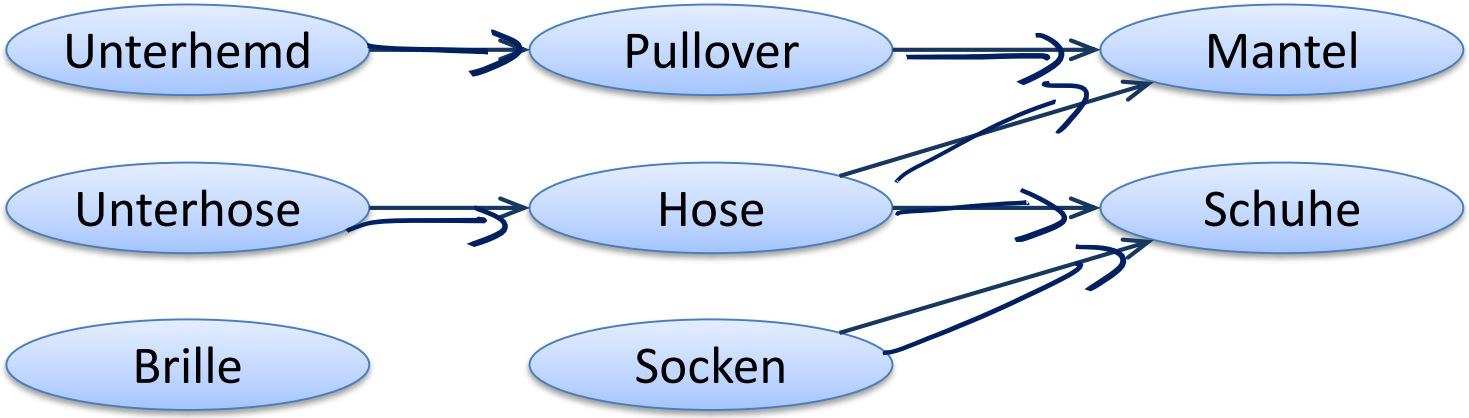
⇒ es gibt eine Rückwärtskante

⇒ (v, u) ist Rückwärtskante

Anwendung: Topologische Sortierung

Zyklenfreie, gerichtete Graphen:

- **DAG**: directed acyclic graph
- Modellieren z.B. zeitliche Abhängigkeiten von Aufgaben
- Beispiel: Anziehen von Kleidungsstücken



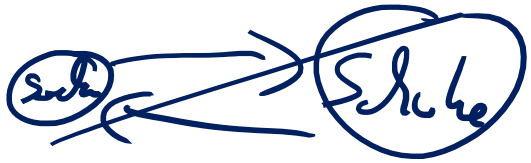
Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass u vor v erscheint, falls ein gerichteter Pfad von u nach v existiert
- Im Beispiel: Finde eine mögliche Anziehreihenfolge

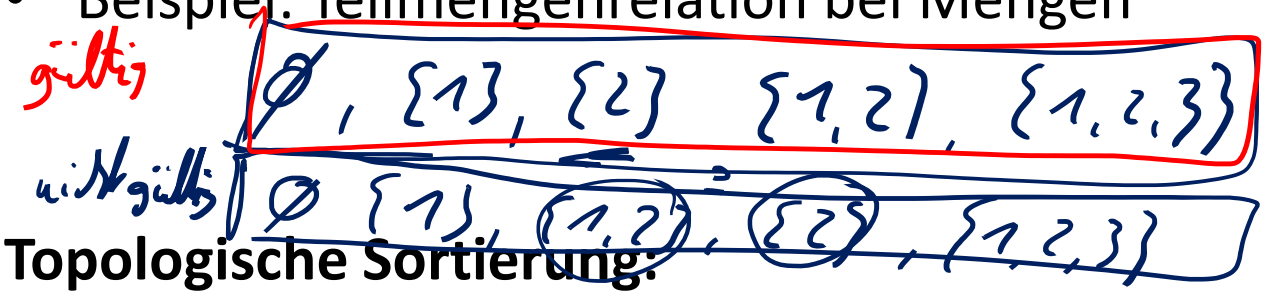
Topologische Sortierung: Etwas formaler…

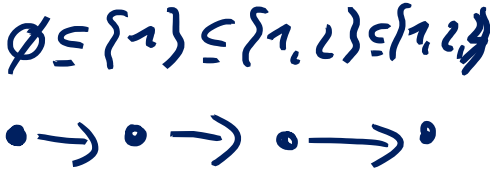
Zyklenfreie, gerichtete Graphen:

- repräsentieren partielle Ordnungsrelationen
  - asymmetrisch: $a < b \implies \neg(b < a)$
    transitiv: $a < b \wedge b < c \implies a < c$
  - partielle Ordnung: nicht alle Paare müssen vergleichbar sein



- Beispiel: Teilmengenrelation bei Mengen



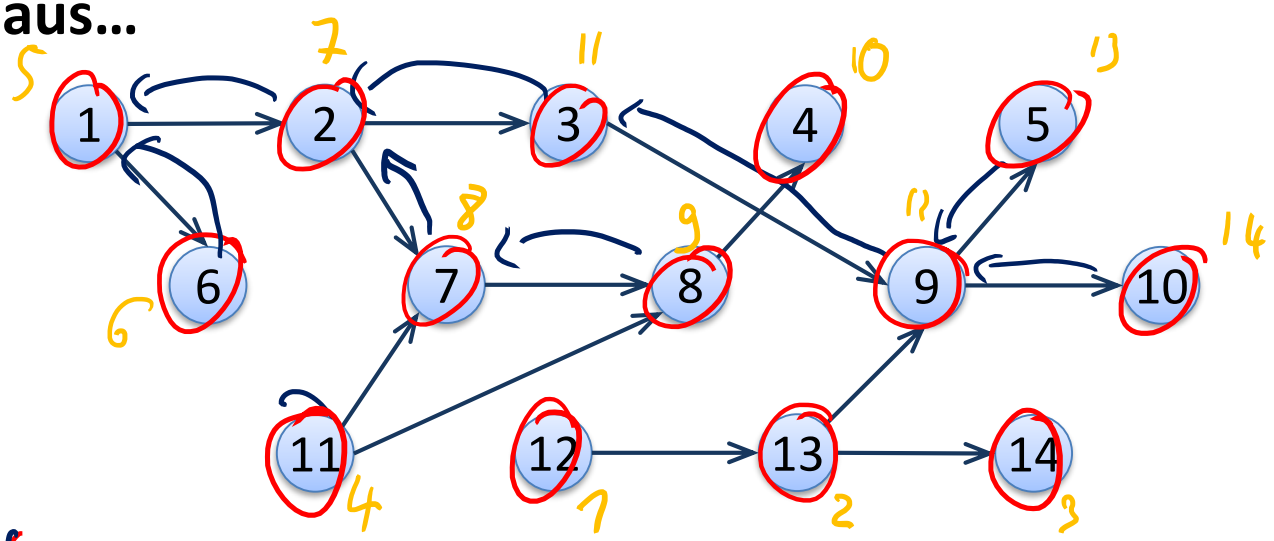


Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass u vor v erscheint, falls ein gerichteter Pfad von u nach v existiert
- Erweitere eine partielle Ordnung zu einer totalen Ordnung

Topologische Sortierung: Algorithmus

Führe DFS aus…



$t_{1,1}, t_{2,1}, t_{3,1}, t_{9,1}, t_{10,1}, t_{10,2}, t_{5,1}, t_{5,2}, t_{9,2}, t_{3,2}, t_{7,1}, t_{8,1}, t_{4,1}, t_{4,2}$

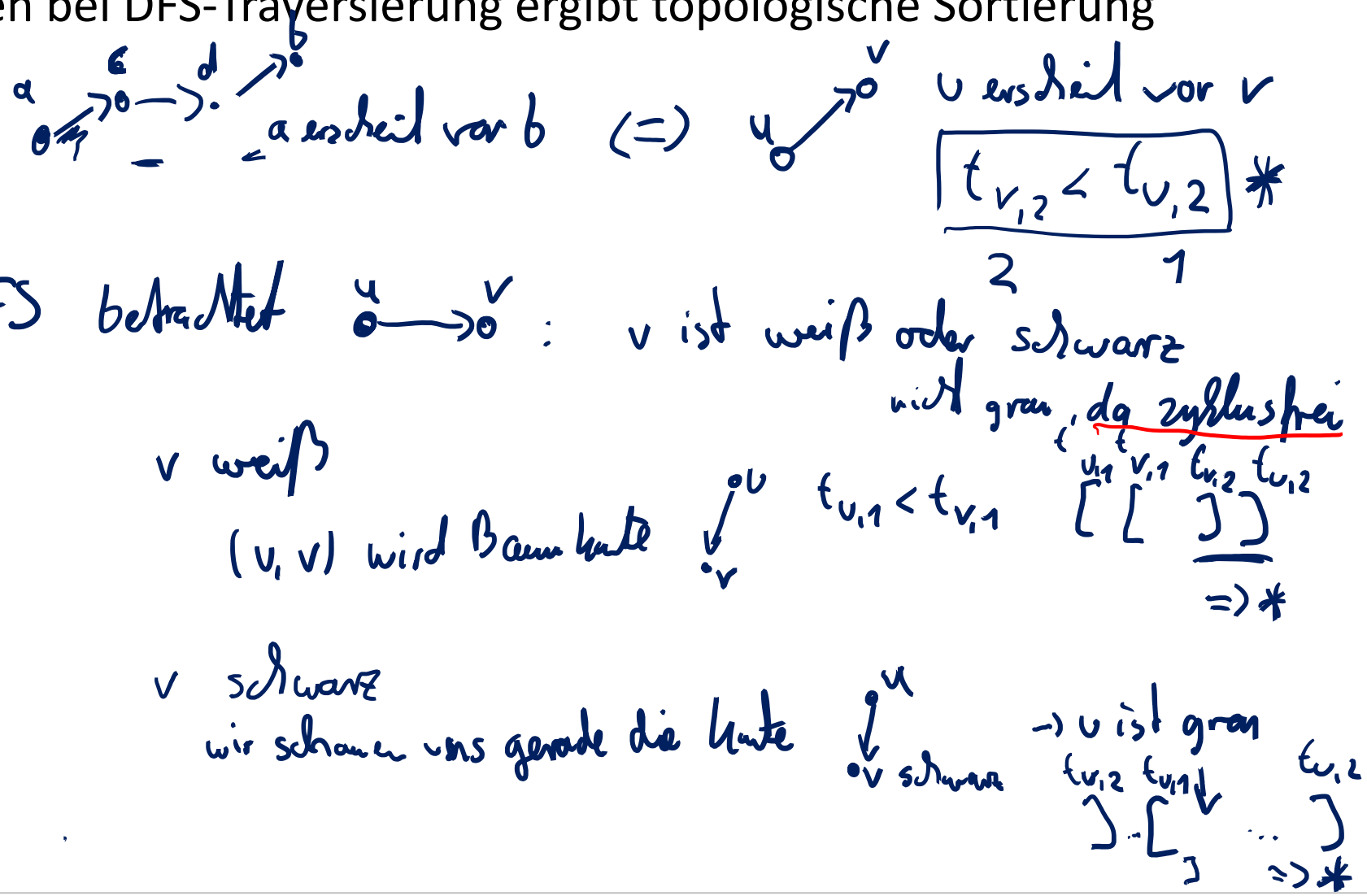
$t_{8,2}, t_{7,2}, t_{2,2}, t_{6,1}, t_{6,2}, t_{11,1}, t_{11,2}, t_{12,1}, t_{13,1}, t_{14,1}, t_{14,2}, t_{13,2}, t_{12,2}$

Beobachtung:

- Knoten ohne Nachfolger werden als erstes besucht (schwarz gef.)
- Besuchreihenfolge ist umgekehrte topologische Sortierung

Topologische Sortierung: Algorithmus

Theorem: Umgekehrte "Visit"-Reihenfolge (schwarz färben) der Knoten bei DFS-Traversierung ergibt topologische Sortierung



Topologische Sortierung: Algorithmus

Theorem: Umgekehrte “Visit”-Reihenfolge (schwarz färben) der Knoten bei DFS-Traversierung ergibt topologische Sortierung

DFS-Traversierung: Weitere Anwendungen

Stark zusammenhängende Komponenten

- Stark zus.-hängende Komponente eines gerichteten Graphen: "Maximale Knoten-Teilmenge, so dass jeder jeden erreichen kann"

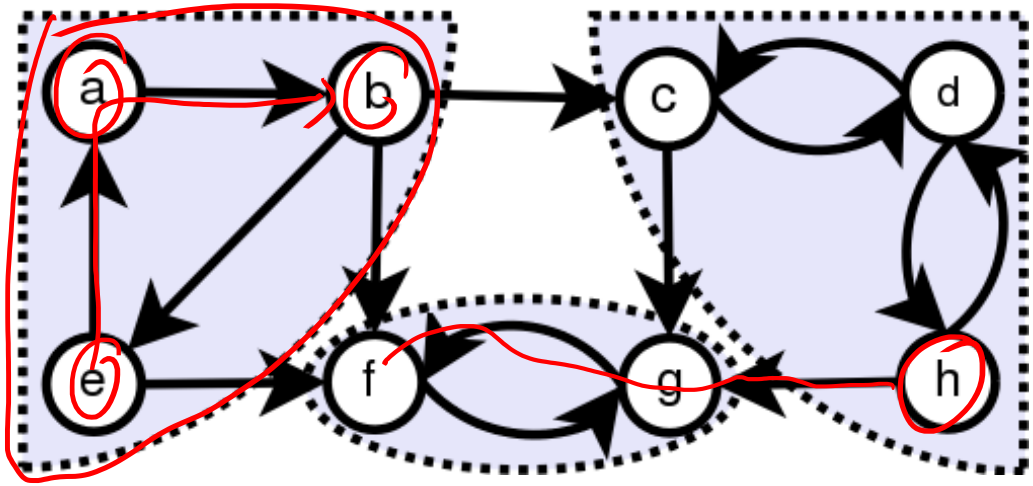


Bild: Wikipedia

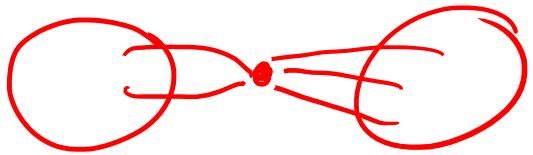
- Benötigt 2 DFS-Traversierungen (Zeit $\in O(m+n)$)
  - auf G und auf G^T (alle Kanten umgedreht)
  - G und G^T haben die gleichen stark zus.-hängenden Komponenten
- Details z.B. in [CLRS]

DFS-Traversierung: Weitere Anwendungen

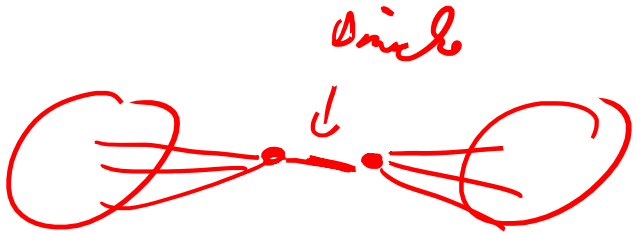
UNI FREIBURG

Artikulationsknoten, Brücken, Biconnected Components

- Annahme: ungerichteter Graph
- **Artikulationsknoten v:**
  v entfernen vergrössert die Anzahl Komponenten



- **Brücke e:**
  Kante e entfernen vergrössert die Anzahl Komponenten



DFS-Traversierung: Weitere Anwendungen

Biconnected Components

- Komponenten, welche übrig bleiben, wenn man alle Brücken entfernt

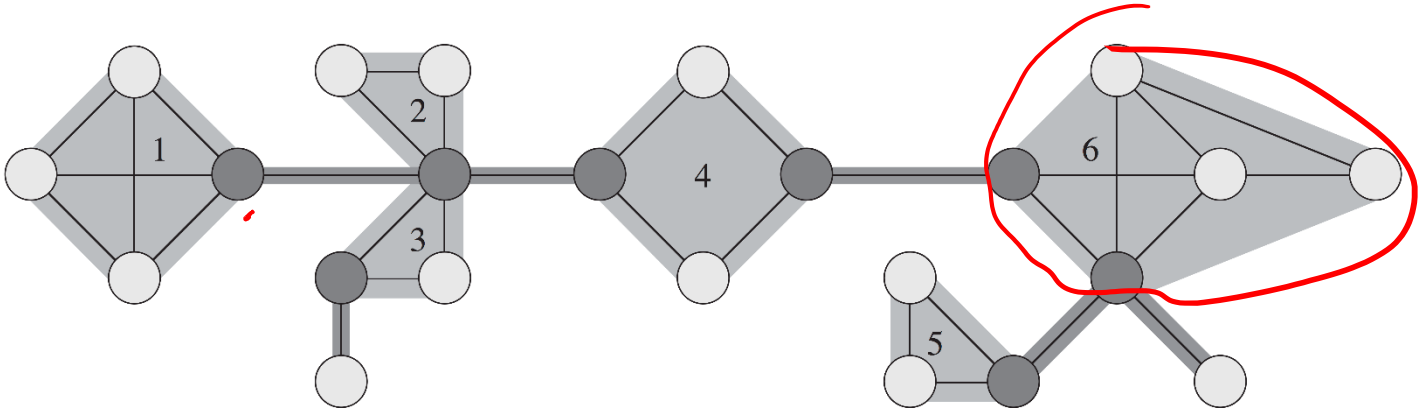


Bild: [CLRS]

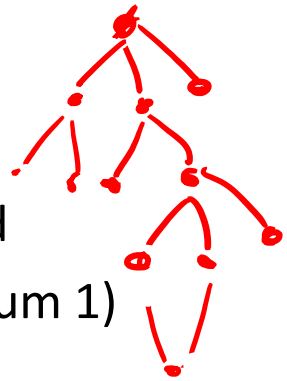
- Artikulationsknoten und Brücken können mit einer DFS-Traversierung in $O(m+n)$ Zeit gefunden werden
  - Algorithmus von Hopcroft, Tarjan (1973)
- Zerlegung in Biconnected Components daher in der gleichen Zeit

Bäume

- Als ungerichtete Graphen (mit n Knoten) betrachtet…

Baum:

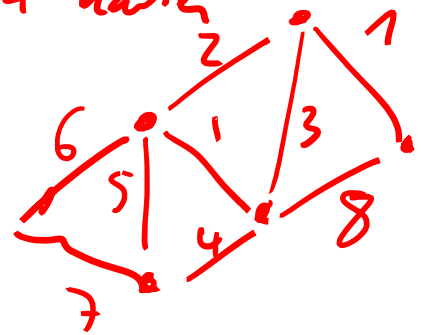
- Zusammenhängender ungerichteter Graph, ohne Zyklen
  - Ein nicht zus.-hängender zyklenfreier (unger.) Graph heisst Wald
  - Anzahl Kanten: $n-1$ (jede Kante reduziert die #Komponenten um 1)



Wald: mehrere Bäume

n − # Komponente = # Kante

$n-4$ Kante



Äquivalente Definitionen:

- Minimaler zusammenhängender Graph
- Maximaler zyklenfreier Graph
- Eindeutiger Pfad zwischen jedem Knotenpaar
- Zusammenhängender Graph mit $n-1$ Kanten